

# Visual Programming and Debugging for Parallel Computing

James C. Browne and Syed I. Hyder  
University of Texas at Austin

Jack Dongarra  
University of Tennessee at Knoxville and Oak Ridge National Laboratory

Keith Moore and Peter Newton  
University of Tennessee at Knoxville

*/// Annotated directed-graph representations of parallel programs simplify programming and debugging by providing a single, consistent framework that separates a program's sequential computations from its parallel structure.*

Parallel architectures have clearly emerged as the future environments for high-performance computation for most applications. The barrier to their widespread use is that writing parallel programs that are both efficient and portable is quite difficult. Parallel programming is more difficult than sequential programming because parallel programs must express not only the sequential computations, but also the interactions (communication and synchronization) among those computations that define the parallelism. To achieve good performance, programmers must understand this large-scale structure.

Most current text-based parallel-programming languages either implicitly define parallel structure, thus requiring advanced compilation techniques, or embed communication and synchronization with sequential computation, thus making program structure difficult to understand. Furthermore, direct use of vendor-supplied procedural primitives can preclude portability. We could alleviate these difficulties with a programming process that separates specification of sequential computations from specification of synchronization and communication, and expresses synchronization and communication directly but abstractly.

Directed-graph program representations can separate these specifications by permitting a two-step programming process in which programmers first design sequential components and then compose them into a parallel structure. This provides a simplified divide-and-conquer approach to design.

Such program representations have other advantages. They directly represent multiple threads of control. They also display and expose the large-scale program structure that parallel programmers must understand

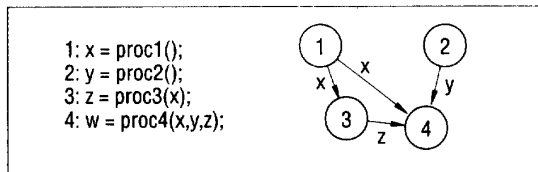


Figure 1. Dataflow example: (a) a sequence of assignment statements, (b) an equivalent dataflow graph. Each circle represents an assignment statement.

to achieve good performance, and a graph model can provide the basis for a visual programming environment in which programming, logical debugging, and performance debugging are integrated into a single common framework.

These are not new ideas; many optimizing compilers already generate directed-graph representations of parallel programs, and performance monitoring systems use graphical displays. The conventional wisdom is that graphical displays of parallel structure are beneficial; our conclusions are natural extensions of this wisdom.

In this article we'll discuss visual parallel programming and how it is implemented in two integrated programming environments—Computationally Oriented Display Environment<sup>1</sup> (CODE) and Heterogeneous Network Computing Environment<sup>2</sup> (Hence)—that represent parallel programs as directed graphs.

### Graph representations of parallelism

The current generation of parallel architectures tends to focus on the MIMD model of parallel computation. MIMD programs are inherently nonlinear, regardless of how they are expressed, because parallel programs consist of multiple interacting threads of control. Directed graphs provide a simple and natural mechanism for representing these multiple threads and helping users understand them.

Consider the sequence of assignment statements in Figure 1a. A parallelizing compiler could analyze the statements, determine that the procedures called cause no dependences, and execute some of the statements in parallel, but the textual representation does not make clear at a glance what the resulting parallel structure will be. The equivalent dataflow graph does (see Figure 1b). Each circle in the graph represents one of the statements. A statement can execute when it has received an input on each incoming arc.

Parallel programs expressed textually are also easier to understand when viewed as a graph. One common means of writing parallel programs is to insert calls to message-passing library routines into otherwise conventional sequential code. Several processes are run in parallel, and they interact through these messages.

The parallel structure of such programs can be diffi-

cult to understand because it is determined by message-passing calls that can be deeply embedded in nested control-flow constructs such as *If* and *While* statements. It can be hard to know precisely under which conditions a given message-passing call will be executed. This problem is commonly addressed by packages that animate the parallel program's execution, often as a space-time diagram with a horizontal bar for each process, and arcs that are drawn from bar to bar to represent message sends. This diagram can be converted into a directed graph simply by converting each bar into a sequence of nodes, dividing the bar at points where messages are sent or received.

Such animations are useful and popular, but the programmer still must relate the space-time diagram back to the program's original textual representation. Bridging this gap between program representation and behavior can be nontrivial.

### Visual parallel programming

If directed graphs are a natural mechanism for displaying the behavior of parallel programs, then why not use them as a basis for a programming language, to reduce the distance between representation and behavior? There are many ways to do this, but we'll discuss a method where the directed graph represents a program's overall parallel structure, and graph nodes with specific icons represent sequential computations.

#### TWO-STEP PROGRAMMING

One immediate advantage of this view is that we can divide the process of creating a parallel program into two steps: creation of components, and composition of these components into a graph. The primitive components can be sequential computations, but other cases are allowed. For example, a component could be a call to another graph that specifies a parallel subcomputation. In any case, components can be either created from scratch or obtained from libraries. The key is that each component maps some inputs to some outputs with a clean and clearly defined interface. These components can then be composed into a graph that shows which components can run in parallel with which other components.

Component creation and component composition are distinct operations. Programmers need not think about the details of one while performing the other (except to ensure that the sequential routines are defined with clean interfaces and well-specified I/O semantics). In particular, programmers can specify the parallel struc-

ture without concern about the inner workings of the components. Furthermore, they can use the best tools available for the different programming tasks.

#### *Sequential components*

Both Hence and CODE emphasize the use of sequential subroutines in C or Fortran as primitive components; in fact, Hence requires it. This facilitates implementation because we build on the existing tool base of tested and accepted sequential languages and compilers. Also, subroutines from existing sequential programs can be incorporated into new parallel programs; leveraging existing code is often vital to the acceptance of new tools. The learning curve for users is less steep because they are not asked to relearn sequential programming when adopting a parallel-programming environment.

#### *Parallel composition into directed graphs*

When designing parallel programs, programmers commonly draw informal diagrams that show large-scale parallel structure. These diagrams abstract away the details of the components being designed and concentrate on their interactions. A graph-based visual parallel programming language can help formalize this process.

Understanding the large-scale structure of parallel programs tends to be more important than for sequential programs, because large-scale structure can dramatically affect the execution performance of parallel programs. For programmers to understand and achieve good performance, they must understand the structure of their program's computation graph—regardless of how their program is represented. For example, if the execution time of sequential segments between communications is too short, performance will suffer because it will be dominated by message-passing overhead.

A direct graphical representation of parallel programs renders such concerns explicit. The programmer knows exactly what the sequential components are because they are separate components. Especially if they are subprograms that perform some cleanly defined function, the programmer will also have a good feel for their execution time. So, the programmer will be aware of the computation's granularity.

The graph can also directly display other information

that is vital to understanding the performance of any parallel program. Issues such as poor load balance or inadequate degrees of parallelism are apparent from the shape of the graph and the execution times of the nodes, interpreted relative to communication overheads.

A graphical representation can also promote locality in designs to the extent to which components are in different name spaces in the language. In CODE, state is retained from one execution of a node to another, and communications must be explicitly defined as part of the interface to a sequential computation node. This encourages programmers to package a node's data with the node. Locality is easy to express, but remote access requires more effort. So, beginning parallel programmers are guided toward designs that exploit good data locality.

#### *CODE and Hence*

One of the challenges in visual programming research is evaluating new ideas. We have found that it is necessary to actually implement systems and use them in programming projects and university programming classes to thoroughly understand

their merits and limits. Even then, results are often subjective and context-dependent. We implemented CODE 2.0 and Hence 2.0, which are based on the ideas described previously. They are similar in purpose and philosophy but are significantly different in detail.

In both languages, users create a parallel program by drawing and then annotating a directed graph that shows the parallel program's structure. Both languages offer several different node types, each with its own icon and purpose. In both cases, the fundamental node type is the sequential computation node, which is represented by a circle icon. The annotations include sequential subroutines that define the computation that the computation nodes will perform, and include the specification of what data the computations will act upon.

The CODE environment runs on Sun 4 workstations and can produce parallel programs for either the Sequent Symmetry or networks of workstations using the PVM portable message-passing library.<sup>3</sup> Hence runs under X Windows on a variety of Unix workstations. Programs that it produces run under PVM on a network of Unix machines of various types and capabilities.

**Directed graphs are a natural way to display the behavior of parallel programs, so we use them as a basis for a programming language, to reduce the distance between representation and behavior.**

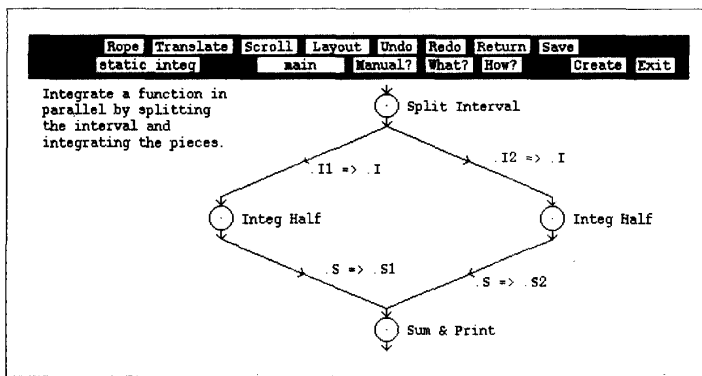


Figure 2. CODE integration program.

## CODE

Figure 2 shows an extremely simple CODE program. It numerically integrates a function in parallel over a definite interval  $[a,b]$  by computing the midpoint  $m$  between  $a$  and  $b$  and then having one sequential computation node integrate the interval  $[a,m]$  while the other does  $[m,b]$  at the same time. The results are summed to form the final result.

The nodes that do the integration are both named `Integ Half`; the graph shows that they can run in parallel because there is no path from one to the other. The arcs represent dataflow from one node to another on FIFO queues. The graph is read from top to bottom, following the arrows on arcs. The graph shows that the `Split Interval` node creates some data that are passed to the two `Integ Half` nodes. The data consist of a structure defining the integration that the receiving node will perform.

To create this parallel program, the programmer uses a mouse to draw a graph just like in Figure 2, and then enters textual annotations into different pop-up windows associated with such objects as nodes and arcs. This information includes such familiar items as type definitions and sequential function prototypes (for type-checking calls). We will ignore these and focus on the annotations of computation nodes. When annotation is complete, the user picks "translate" from a menu, and a parallel program is created, complete with a makefile, ready to be built and run on the selected parallel machine.

### Annotations

The annotation for a computation node consists mostly of a sequence of stanzas, some of which are optional. For example, the annotation for either `Integ Half` node is

```
input_ports { IntegInfo I; }
output_ports { real S; }
vars { IntegInfo i; real val; }
firing_rules { I -> i => }
comp { val = simp(i.a, i.b, i.n); }
routing_rules { TRUE => S <- val; }
```

(Both nodes are identical. A single dynamically replicated node could be used instead.)

The first two stanzas provide names for "ports," which are queues of data that enter and leave the node. Each node uses its own local names for these ports so that nodes can be reused in new contexts. This node will read data of type `IntegInfo` (the structure that defines the work to be done) from port `I`, and write

real data onto port `S`.

Each arc annotation (see Figure 2) binds an output port name to an input port name. The graph clearly shows that the `Split Interval` node places data onto output ports `I1` and `I2`. Port `I1` is bound to input port `I` of the left `Integ Half` node. So, the data that `Split Interval` places onto `I1` is sent to the left `Integ Half` node, and the data placed onto `I2` is sent to the other.

The `vars` stanza of the annotation of `Integ Half` defines variables that are local to the node and that its computation can read and modify.

The `firing_rules` stanza is very important. It defines conditions under which the node can execute. It also describes which local variables will have data placed in them that have been removed from designated ports. CODE's notation for firing rules is quite flexible, but is sometimes complicated relative to the language's other features. The rule "`I -> i =>`" is the simplest case. It signifies that the node can fire when there is data waiting on port `I`, and that when the node fires, the data value is removed from `I` and placed in local variable `i`. Thus, the `Integ Half` nodes simply wait for an incoming value. When one appears, they fire and produce an output.

The `comp` stanza defines what sequential computation will be performed when the node fires. The text is expressed in a language that is a subset of sequential C functions and that includes calls to externally defined sequential functions and procedures (such as `simp`, which does the integration in this example). It is expected, but not required, that all significant sequential computations will be encapsulated in such external functions.

Finally, the `routing_rules` stanza determines what values will be placed onto output ports. As with firing rules, the notation is flexible and potentially complex, but this example is simple. The value of real variable `val` is placed onto queue `S`.

*Other CODE icons*

Figure 3 shows the icons that appear in CODE graphs. Three of them define the interface to a graph. CODE graphs can call other CODE graphs through the Call icon. Arcs incident upon Call nodes are the actual parameters of the call. These arcs are bound to interface nodes in the called graph via a name binding that is an attribute of the arcs. This is similar to arcs binding port names between two nodes, as we described before. Interface nodes must have names that are unique within the graph.

Creation parameters are also bound to an incoming arc. They extract exactly one value from this arc when the called graph is instantiated at runtime. All nodes in the called graph can use the creation parameter name as a constant.

The shared-variable icon declares variables that will be shared among a set of nodes. Each node must declare whether it requires read-only or read-write access to the variable.

**HENCE**

The Hence program in Figure 4 looks exactly the same as its equivalent CODE program in Figure 2, except that

- Hence graphs are read from bottom to top.
- Hence computation nodes are always named by the (exactly) one sequential procedure they are required to call.
- Hence arcs take no annotation.







General nodes	Graph interface definition
 Sequential computation	 Incoming parameter
 Shared-variable declaration	 Outgoing parameter
 Call from one graph to another	 Creation (read-only) parameter

Figure 3. Node icons in CODE.

Figure 4 shows all the node annotations; in the actual Hence system, the annotations are in pop-up windows.

Although the Hence graph looks like the CODE graph, its meaning is very different. Except for some features that have not been discussed, arcs in CODE represent dataflow. Arcs in Hence represent two different concepts at the same time: control flow and variable name scope.

A Hence node can execute whenever all of its predecessors have executed. This is the only rule that defines when a node can run, and there is no implication that predecessor nodes have sent any data. There are no explicit node-firing rules as in CODE. Hence has special control-flow nodes that can alter the succession of node executions.

*Annotations*

Hence node computations access variables in a global name space. Each node must contain declarations that specify which variables will be accessed and whether changes to them will be propagated to successor nodes in the graph. This will require an explanation and some background. Computation node annotations have three parts, two of which are optional:

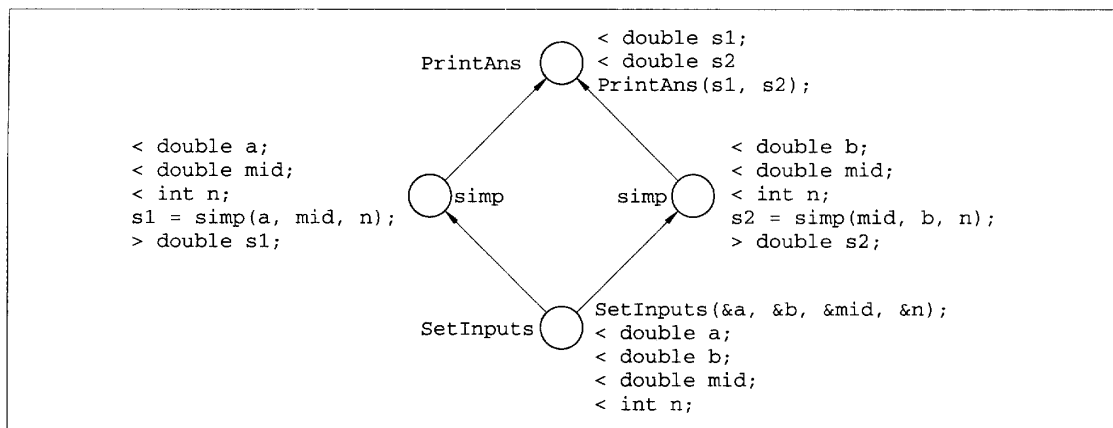


Figure 4. Hence integration program.






	Sequential computation.
	Loop begin and end. The enclosed subgraph is iterated over an index range such as $i = 0$ to $N$ .
	Conditional begin and end. The enclosed subgraph is executed only if an expression evaluates to TRUE.
	Parallel replication (fan) begin and end. The enclosed subgraph is replicated such that all copies execute in parallel. Copies are indexed as in $i = 0$ to $N$ .
	Pipeline begin and end. The enclosed subgraph is replicated to form a pipeline with indexed stages.

Figure 5. Node icons in Hence.

1. *Declaration of input and input/output variables* (optional). The values of input and input-output variables are read from the nearest predecessor node that outputs those variables. The values of the variables can be changed. New values of input-output variables can be seen by successor nodes; new values of input variables cannot. Input declarations contain a "<" token, and input-output declarations contain a "<>" token.
2. *Call to a sequential procedure* (required). The procedure may be written in either C or Fortran. The call's actual parameters may be expressions. Variables that appear in the expressions are inputs, input-outputs, or outputs from the node.
3. *Declaration of output variables* (optional). The node can set output variables, whose values are available to successor nodes. Output declarations contain a ">" token.

Consider the annotation of the `SetInputs` node in Figure 4. It calls a C routine named `SetInputs`, which provides values for variables  $a$ ,  $b$ ,  $mid$ , and  $n$ . The variables are made available to successor nodes because they appear in output declarations.

The two `simp` nodes are very similar, but one uses input declarations to read  $a$ ,  $mid$ , and  $n$  from its nearest predecessor (`SetInputs`) and the other reads  $mid$ ,  $b$ , and  $n$ . The left `simp` node makes  $s1$  available to its successors in the graph, and the right makes  $s2$  available. These variables hold the results of the integration. Subroutine `simp`, a C procedure, actually performs the integration.

The `PrintAns` node reads  $s1$  and  $s2$ . It calls the C procedure `PrintAns`, which sums them and prints their value, which appears in the Hence console window when the program is run.

#### Other Hence icons

Like CODE, the Hence language also supports icons other than the circle that represents a sequential computation. These other node types represent control structures. Hence has no facility for hierarchical implementation, although its model could support it. Hence

graphs cannot call other graphs, so there is no need for interface nodes. Figure 5 shows all of Hence's icons.

Hence's control-flow icons work in pairs; one icon begins a construct and another ends it. The subgraph that appears between the icons is acted upon. For example, the subgraph between a loop-begin and a loop-end node is executed repeatedly, much like the body of a C `for` loop. The loop-begin is annotated with a statement that assigns its index variable an initial value, with a termination condition expression, and with a statement that gives the index variable its next value. The loop-end node (and all other end nodes) requires no annotation.

Conditional node pairs define an "if-then" structure. The conditional-begin node annotation contains an expression. If the expression evaluates to TRUE (meaning nonzero, following the C language convention), the subgraph between the pairs is executed. Otherwise, it is not. Hence does not contain an "if-then-else" structure.

Fan node pairs create parallel structures. They dynamically replicate the subgraph between them and evaluate the replications in parallel. The fan-begin node's annotation consists of an index statement:

```
IndexVar = StartValue TO EndValue;
```

`IndexVar` takes a different value in each replicated subgraph, so each replication has a unique index.

#### CODE AND HENCE COMPARED

Node-firing conditions are explicit and general in CODE. Programmers must explicitly define the exact circumstances under which a computation node can execute. The specification language is quite flexible and general, and firing conditions can depend on the node's internal state. For example, it is easy to define a node that nondeterministically merges data from two streams. Such a computation is impossible to state in Hence.

It is tempting to say that Hence's firing rules are fixed. Nodes can fire when all predecessor nodes have fired, but this is an oversimplification. Execution of a Hence node is dictated also by the control-flow constructs in which it is embedded. So, Hence firing conditions are somewhat less explicit.

CODE can express more dynamic graph topologies. Because of its powerful firing rules and its method of instantiating nodes, CODE can express communications patterns that Hence cannot. For example, CODE can accept an adjacency graph as input data and create a graph with the specified topology.

CODE graphs explicitly show dataflow; Hence graphs show control flow. Control flow, in the tradi-

tional sense that applies to Hence, is expressed declaratively in CODE firing rules.

CODE's basic unit of component reuse is the sequential computation node. The CODE model supports libraries of computation nodes. Thus, CODE computation nodes must be completely encapsulated. They must have well-defined interfaces, and they must be completely defined in isolation from other graph elements. This is why CODE ports exist; they are a node's formal parameters.

Hence nodes are not a natural unit of reuse because of the way they use variable names. A programmer who copies a node from one Hence program to another will likely have to edit the node when it is placed in its new context. For example, the new program may name some array A, whereas the old program called it B. However, the subroutines called from a node can be reused. These contain the bulk of the program text.

CODE allows name binding on arcs, thus bridging the name spaces of any two nodes. The downside is that programmers must specify name bindings on every arc. The CODE computation node is somewhat analogous to a simple statement in a conventional programming language like Fortran. It is cumbersome to have to specify name binding between "statements."

#### LESSONS LEARNED

It is not surprising that there is a trade-off between ease of use and expressive power. CODE's firing rules are explicit and general, but can be complicated and wordy. Hence's firing rules are simple and concise, but are not always adequate to express algorithms. They do appear to be adequate for many interesting numerical algorithms, however. Languages should be designed with simple mechanisms to cover 90% of the needs but should also include more expressive mechanisms. The challenge is to define both so that they compose naturally.

Parallel structure is often not fully determined until runtime. CODE and Hence address this by using dynamic replications, but this reduces the degree to which the program's structure is visually apparent. The static display of programs whose structure is determined at runtime remains a significant goal.

CODE shows all dataflow or common-shared-variable

access via arcs. This more completely shows the communication patterns of programs, but dataflow graphs are often complex. Programs with complex dataflow can become a rat's nest of arcs. This can be hard to understand and is also cumbersome because programmers must individually annotate all arcs.

Hence programs are related to flow charts of structured programs. They are concise and orderly even when graphs become large. Dataflow is implicit, so less structural information is presented to the programmer; but

computational elements are still clear and well encapsulated, and parallel structure is displayed. However, because dataflow is implicit, programmers might make errors in which the "wrong" node is another node's nearest predecessor for some variable.

**With visual parallel programming languages, performance and logical debugging can be carried out with the same representation used for programming.**

#### *Debugging in the visual environment*

Both performance and logical debugging are important aspects of parallel programming. Performance debugging involves understanding and improving a program's speed, and logical debugging involves correcting errors in a program's logic. With

visual parallel programming languages, performance and logical debugging can be carried out with the same representation used for programming. This article focuses on logical debugging.

#### DEBUGGING ENTITIES

Debugging establishes the relationship between the program (typically some small segment of a large program) and its execution. The entities involved in debugging include the program, *P*; a specification of the program's (or program segment's) expected execution behavior, which we call *M* for model of behavior; and some representation of the program's actual execution behavior, *E*.

Debuggers for programs written in pure text forms typically use different representations for *P*, *M*, and *E*, and this imposes much additional work on the programmer. Ideally, these entities would be expressed in the same representation, so that the programmer will not have to understand and manipulate several different notations.

Visual directed-graph programming supports the formulation of parallel-program debuggers in which a single notation can express all the entities. This simplifies

debugging not only because it lets programmers think in terms of the program, which is what they understand, but also because it allows the ready automation of the often tedious task of comparing expected and actual behaviors. It also facilitates identification of the program's logic faults. We are implementing such a debugger for the CODE system.<sup>4</sup>

#### DEBUGGING THE GRAPH

An *action* is a node's atomic execution. It maps an input state into an output state according to a known I/O relation. The execution of a program is the traversal of the graph, starting with an assignment of an initial state, and ending with the execution of a final state. The traversal causes *events*, which are the execution of the actions at the nodes, and generates a partially ordered set of events. This set defines a directed acyclic graph of events that corresponds to instantiations of the actions defined at the nodes.

Debugging identifies the actions responsible for the program's failure to meet its final state specification. Debugging comprises these steps:

1. *Identify and select the portions of the program whose behavior will be monitored.* This is a set of "suspect" nodes or subgraphs. It is typically impossible to monitor the entire execution behavior of large, complex parallel programs. The visual graph representation of  $P$  simplifies selection of suspect portions.
2. *Specify the expected execution behavior of the set of nodes that will be monitored.* A graphical program's execution behavior is naturally represented as the partially ordered set of events expected to be generated at the nodes of the suspect subgraphs. This is  $M$ , the model of expected behavior. We can either construct this set of events directly, or construct a graph of the actions whose execution will generate the desired set.
3. *Capture the execution behavior of the selected portions of the program.* The execution behavior is the partially ordered sequence of events that actually occurred during execution. This set is  $E$ . The programmer obtains  $E$  by selecting a set of program nodes with the mouse and then running the program. The system then automatically records all the necessary events and orderings.

**A visual programming environment provides a unified framework for supporting different concurrent debugging facilities.**

4. *Map  $E$  to  $M$  to determine where the actual and expected events first diverge.* This can be done automatically because  $E$  and  $M$  are specified in the same representation. The mapping identifies event sequences in  $E$  that do not correspond to the allowed set defined in  $M$ .
5. *Map the elaborated graph of  $M$  back to  $P$  to define corrective action.* Because the elaborated graph of  $M$  contains instances of the nodes of  $P$ , the mapping is automatic and guides us toward the offending action in  $P$ .

A programmer will often cycle through these steps a number of times before identifying the bug. In each cycle, the programmer will progressively come closer to the offending piece of code.

Using actions, instead of events, in the representation of  $M$  greatly helps the debugger filter out irrelevant information. This restricts the execution history displays of  $E$  to only the events that interest the user. This filtering greatly simplifies the checking of  $M$ , which can be done either visually by the programmer with the help of the debugger's execution displays, or automatically by

the debugger's model-checking facility. The debugger provides an animation facility that visualizes the mapping of  $E$  to  $M$ . The elaborated graph acts as an underlying structure that greatly helps the animation.

So, a visual programming environment provides a consistent graphical representation for all the entities used in debugging and simplifies the design of a concurrent debugger that coherently relates the steps of debugging. It also provides a unified framework for supporting different concurrent debugging facilities, like execution history displays, animation, and model-checking facilities.

**C**ODE and Hence have demonstrated many advantages of visual programming for parallel computing. A visual program representation that directly shows program structure allows programming, debugging, and performance analysis in a single consistent framework. It also allows the programming process to be separated into two phases: the creation of sequential components, and their composition into a parallel program.



Only by actually implementing the programming environments have we been able to evaluate these benefits and discover even more effective visual parallel-programming abstractions. Much work remains to be done. Even with visual techniques, it is difficult to express statically those program structures that are largely determined at runtime. Also, compilation techniques could be improved because CODE and Hence lend themselves to automatic dataflow and scheduling analysis.

There are also alternative methods for defining the primitive compute-node computations. The University of Tennessee is developing a visual programming environment called VPE,<sup>5</sup> which features compute nodes in which users place explicit message-passing calls. This can simplify the computation-graph structure of many programs.

#### FURTHER INFORMATION

Hence and the PVM package it targets are available in source and binary form via xnetlib or by anonymous ftp to ftp.netlib.org. E-mail questions on Hence to hence@cs.utk.edu. For more information on Code, contact browne@cs.utexas.edu or newton@cs.utk.edu. A World Wide Web site for information on PVM/Hence is at [http://www.epm.ornl.gov/pvm/pvm\\_home.html](http://www.epm.ornl.gov/pvm/pvm_home.html).

#### REFERENCES

1. P. Newton, "A Graphical Retargetable Parallel Programming Environment and Its Efficient Implementation," Tech. Report TR93-28, Dept. of Computer Sciences, Univ. of Texas, Austin, Tex., 1993.
2. A. Beguelin et al., "Visualization and Debugging in a Heterogeneous Environment," *Computer*, Vol. 26, No. 6, June 1993, pp. 88-95.
3. G.A. Geist et al., "PVM 3 User's Guide and Reference Manual," Tech. Report ORNL/TM-12187, Oak Ridge Nat'l Laboratory, Oak Ridge, Tenn., 1993.
4. S.I. Hyder, J.F. Werth, and J.C. Browne, "A Unified Model for Concurrent Debugging," *Proc. Int'l Conf. Parallel Processing, Vol. 2: Software*, IEEE Computer Society Press, Los Alamitos, Calif., 1993, pp. 58-67.
5. P. Newton and J. Dongarra, "Overview of VPE: A Visual Environment for Message-Passing Parallel Programming," Tech. Report UT-CS-94-261, Computer Science Dept., Univ. of Tennessee, Knoxville, Tenn., 1994.

**James C. Browne** holds a Regents Chair in Computer Sciences at the University of Texas at Austin. Parallel programming is his most active and current research interest. His previous research areas include operating systems, and quantum mechanical calculations of small molecules. He received his PhD in physical chemistry from the University of Texas at Austin in 1960, and his BA from Hendrix College, Conway, Arkansas, in 1956. He can be contacted at the Dept. of Computer Sciences, Univ. of Texas at Austin, Austin, TX 78712-1188; Internet: browne@cs.utexas.edu.

**Syed Irfan Hyder** earned his PhD and MS, both in electrical and computer engineering, from the University of Texas at Austin in 1994 and 1989. He earned his MBA in 1987 from the Institute of Business Administration at Karachi University, Pakistan. He received his BE in electrical engineering from the NED University of Engineering and Technology, Karachi, in 1985. He can be contacted at 201/O Block 2, PECHS, Karachi 75400, Pakistan.

**Jack Dongarra** holds a joint appointment as Distinguished Professor of Computer Science in the Computer Science Department at the University of Tennessee and as Distinguished Scientist in the Mathematical Sciences Section at Oak Ridge National Laboratory under the UT/ORNL Science Alliance Program. He specializes in numerical algorithms in linear algebra, parallel computing, use of advanced-computer architectures, programming methodology, and tools for parallel computers such as PVM/Hence and MPI. Other current research involves the development, testing, and documentation of high-quality mathematical software. Dongarra received his PhD in applied mathematics from the University of New Mexico in 1980, his MS in computer science from the Illinois Institute of Technology in 1973, and his BS in mathematics from Chicago State University in 1972. He is a member of the IEEE, the SIAM, and the ACM. He can be contacted at the Dept. of Computer Science, 107 Ayres Hall, Knoxville, TN 37996-1301; Internet: dongarra@cs.utk.edu.

**Keith Moore** is a research associate at the Computer Science Department of the University of Tennessee, Knoxville. He is involved in the development of protocols for very-large-scale, fault-tolerant, networked information retrieval on the Internet. In addition to his work on Hence, he contributed to the design of PVM version 3, and to the MIME standard for multimedia electronic mail. He is a candidate for an MS in computer science at the University of Tennessee, and he received his BS in electrical engineering from Tennessee Technological University in 1984. He is a member of Computer Professionals for Social Responsibility. He can be contacted at the Dept. of Computer Science, Univ. of Tennessee, 107 Ayres Hall, Knoxville, TN 37996-1301; Internet: moore@cs.utk.edu.

**Peter Newton** holds a post-doctoral research position at the Computer Science Department of the University of Tennessee, Knoxville. His interests center on performance analysis and programming environments and languages for parallel computing. He received his PhD and MS in computer science in 1993 and 1988, from the University of Texas at Austin. He earned his BS in mathematics and computer science from the University of Michigan in 1982. He is a member of the IEEE Computer Society and the ACM. He can be contacted at the Dept. of Computer Science, Univ. of Tennessee, 107 Ayres Hall, Knoxville, TN 37996-1301; Internet: newton@cs.utk.edu.